UNIVERSITÉ
CÔTE D'AZUR

Université Côte d'Azur — L2 Informatique
Programmation C

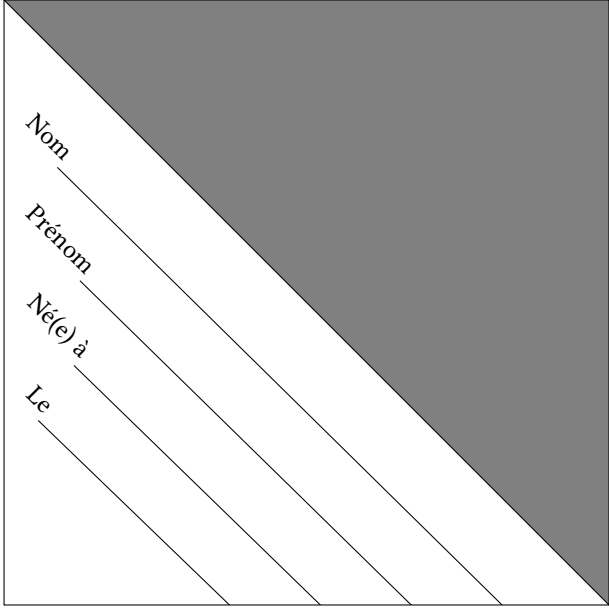
Examen : Programmation C

23 mai 2025

Durée : 2 heures
Aucun documents autorisés. Il est interdit d'accéder à internet.

Note

Toutes les questions sont indépendantes.
Tous les codes devront être écrits en **Langage C ANSI**. La notation est donnée à titre indicatif.
Nombre de pages : 8



Ne surtout pas rabattre le triangle grisé. Il est là simplement pour faire sérieux. Dans tous les cas, votre copie ne sera pas anonyme car c'est le même enseignant qui corrige et qui rentre les notes.

Exercice 1 : Chaînes de caractères (3 points)

Dans cet exercice et les suivants, il est interdit d'utiliser les fonctions de la bibliothèque standard, en particulier la bibliothèque `string.h` (`strcat`, `strcpy`, `strcmp`, etc). Pour la bibliothèque `stdlib.h`, seules les fonctions `malloc` et `free` sont autorisées. Des points seront enlevés à ceux ne respectant pas la norme ANSI.

1. Écrire une fonction `int length(char *chaine)` qui calcule et renvoie la longueur de la chaîne donnée en paramètre.

```c
int length(char *chaine) {
    int i;
    for (i=0; chaine[i] != '\0'; i++);
    return i;
}
```

2. Écrire une fonction `int equal(char *chaine1, char *chaine2)` qui teste l'égalité de deux chaînes. On renverra une valeur booléenne sous forme d'entier.

```c
int equal(char *chaine1, char *chaine2) {
    int i;
    for (i=0 ; chaine1[i] != '\0'; i++) {
        if (chaine1[i] != chaine2[i]) return 0;
    }
    if (chaine2[i] == '\0') return 0;
    return 1;
}
```

3. Écrire une fonction `char *copy(char *chaine)` qui renvoie une nouvelle chaîne allouée sur le tas, dont le contenu est identique à la chaîne donnée en paramètre.

```
char *copy(char *s) {
    char *new = malloc(length(s) + 1);
    for (i=0 ; s[i] != '\0'; i++) {
        new[i] = s[i];
    }
    new[i] = '\0';
    return new
}
```

Exercice 2 : Messages secrets (3 points)

Deux agents secrets souhaitent communiquer ensemble. Leurs messages se devant être courts, ils sont codés avec des *shorts* (le type C sur 16 bits et non le vêtement). Chaque message est constitué de trois éléments :

1. un caractère sur 7 bits codant pour l'expéditeur (*exemple : `'A'` = 65 = 100 0001$_b$ pour représenter Alice*)
2. un autre caractère sur 7 bits codant pour le destinataire (*exemple : `'B'` = 66 = 100 0010$_b$ pour Bob*)
3. un booléen avancé (`enum advanced_boolean`) codé sur 2 bits (*exemple 01 pour représenter un NON*).

Ainsi `'A'+'B'+NON` sera encodé par l'entier 33545 dont la représentation binaire est :

100 0001	100 0010	01

```
enum advanced_boolean {
    OUI,       /* Affirmatif   */
    NON,       /* Négatif      */
    BOF,       /* Dubitatif    */
    HEIN_QUOI /* Interrogatif */
};
```

1. Écrire les fonctions `char first_char(short message)` et `char second_char(short message)` qui renvoient respectivement le premier et second caractère du message. Chaque fonction ne devra contenir qu'un calcul d'une seule ligne.

```
char first_char(short message) {
    return (message>>2) & ((1<<7) - 1);
}
char second_char(short message) {
    return (message>>9) & ((1<<7) - 1);
}
```

2. Écrire la fonction `enum advanced_boolean boolean(short message)` qui renvoie le booléen avancé associé au message. La fonction doit être écrite en une seule ligne.

```
enum advanced_boolean boolean(short message) {
    return message & 3; /* 3 = 11b */
}
```

3. Écrire la fonction `short fast_answer(short message)` qui construit un message en partant du paramètre et qui inverse les deux caractères en gardant le même booléen avancé. Exemples : si `message` code pour `'F'+'D'+BOF` alors la fonction renverra le nombre dont la représentation binaire correspond à `'D'+'F'+BOF`

```
short fast_answer(short message) {
    char c1 = first_char(message);
    char c2 = second_char(message);
    enum advanced_boolean b = boolean(message);
    return (c2 << 9) + (c1 << 2) + b;
}
```

Exercice 3 : Dictionnaire (8 points)

Dans cet exercice, on apportera un soin tout particulier à la gestion mémoire. En particulier, il faudra vérifier deux invariants :

1. Chaque pointeur stocké dans notre dictionnaire doit être unique. Ainsi, on peut le libérer lorsqu'il n'est plus utile sans avoir de risque de le voir utilisé à un autre endroit de notre programme. En particulier, cela signifie qu'à chaque fois que l'on voudra transférer des données vers notre dictionnaire ou depuis notre dictionnaire, on fera attention de copier les valeurs (*cf* exercice 1 dont on pourra utiliser les fonctions).
2. Après l'appel de la fonction `free_dict`, toutes les allocations (directes ou indirectes) faites par les fonctions de cet exercice devront avoir été libérées. Attention, il est possible que certaines libérations doivent avoir lieu dans d'autres fonctions. C'est à vous de vous assurer qu'il n'y a pas de fuite mémoire.

Objectifs

On souhaite implémenter un dictionnaire, c'est-à-dire, une structure associant des clés à des valeurs. En python, cela se noterait `dico["clef"] = "une valeur"`. Dans cet exercice, les clés et les valeurs seront toutes deux des chaînes de caractères. Une association entre une clé et une valeur sera appelée `map`. Notre dictionnaire sera un tableau dynamique d'association, c'est-à-dire de `map`.

1. Définir une structure `struct map` (que l'on renommera simplement `map`) contenant deux champs de type chaînes de caractères : un champ `key` et un champ `value`. Écrire aussi la fonction `map new_map(char *k, char *v)` qui renvoie une nouvelle `map` à partir d'une clé `k` et d'une valeur `v`. On fera attention que les chaînes stockées dans la `map` soient indépendantes en mémoire des chaînes données en paramètre (même contenu mais adresses distinctes).

```
typedef struct map {
    char *key;
    char *value;
} map;

map new_map(char *k, char *v) {
    map m;
    m.key = copier(k);
    m.value = copier(v);
    return m;
}
```

2. Définir maintenant la structure `struct dict` (renommé en `dict`) correspondant à un dictionnaire. Cette dernière devra avoir trois champs : un tableau `array` contenant des objets de type `map`, un entier `capacity` représentant la taille effective du tableau et un entier `last` indiquant le premier indice libre dans le tableau.

```
typedef struct dict {
    map *array;
    int capacity;
    int last;
} dict;
```

3. Écrire maintenant le constructeur `dict new_dict(int capacity)` qui construit un nouveau dictionnaire à partir de la capacité donnée en paramètre.

```
dict new_dict(int capacity) {
    dict d;
    d.array = malloc(sizeof(map) * capacity);
    d.capacity = capacity;
    d.last = 0;
    return d;
}
```

4. Écrire maintenant `void grow(dict d)` qui double la capacité du dictionnaire tout en conservant le même contenu. On rappelle qu'on ne pourra pas utiliser la fonction `realloc`.

```
void grow(dict d) {
    map *new = malloc(2 * d.capacity * sizeof(map));
    int i;
    for (i=0; i<d.last; i++) {
        new[i] = d.array[i];
    }
    free(d.array);
    d.array = new;
    d.capacity *= 2;
}
```

5. Afin de préparer la question suivante, écrire une fonction `void update_map(map *m, char* value)` qui remplace la valeur de la `map` `m` par celle donnée en paramètre.

```
void update_map(map *m, char* value) {
    free(m->value);
    m->value = copier(value);
}
```

6. Écrire la fonction `int get(dict d, char *k, char **res)` permettant de récupérer une valeur connaissant sa clé. La fonction renverra l'indice du tableau contenant la `map` correspondante et modifiera le paramètre `res` pour y stocker un pointeur vers une chaîne égale à la valeur associée à la clé.

```
int get(dict d, char *k, char **res) {
    int i;
    map m;
    for (i=0; i<d.last; i++) {
        m = d.array[i];
        if (equal(m.key, k)) {
            *res = copier(m.value);
            return i;
        }
    }
    return -1;
}
```

7. Écrire maintenant la fonction pour modifier une entrée du dictionnaire (si la clé est déjà présente) ou la rajouter (si la clé est absente) : `void set(dict d, char *k, char *v)`

```
void set(dict d, char *k, char *v) {
    char * s;
    int i = get(d,k,&s);
    if (i>=0) {
        update_map(&d.array[i], v);
    } else {
        if (d.last == d.capacity) grow(d);

        d.array[d.last++] = new_map(k,v);
    }
}
```

8. Enfin écrire une fonction `void free_dict(dict d)` qui libère toutes les données du dictionnaire allouées sur le tas.

```
void free_dict(dict d) {
    int i;
    for (i=0; i<d.last; i++) {
        free(d.array[i].key);
        free(d.array[i].value);
    }
    free(d.array);
}
```

Exercice 4 : Allocateur et environnement Vaλisp (6 points)

Vous êtes libre d'utiliser les fonctions ci-dessous définie lors des TP. Vous êtes censés savoir ce que font ces fonctions. Pour résumer, les quatre premières (TP 1) permettent d'accéder aux informations de la mémoire organisée sous la forme d'une liste doublement chaînée. La mémoire est un tableau `MEMOIRE_DYNAMIQUE` de blocs (`uint32_t`). Les six fonctions suivantes (TP 2) permettent de manipuler des `sexpr` représentant soit des entiers, soit des couples.

- TP 1 : Allocateur et mémoire dynamique
  - `int suivant(int i)`
  - `int precedent(int i)`
  - `int taille(int i)`
  - `bool est_libre(int i)`
  - `int rechercher_bloc_libre(int taille)` (`taille` en blocs)
  - `void *allocateur_malloc(size_t size)` (`size` en octets)
  - `void allocateur_free(void *ptr)`
- TP 2 : Types et encodage
  - `sexpr new_integer(int i)`
  - `int get_integer(sexp e)`
  - `bool integer_p(sexp e)`
  - `sexpr car(sexpr e)`
  - `sexpr cdr(sexpr e)`
  - `sexpr cons(sexpr e1, sexpr e2)`

1. On se donne un environnement (une liste lisp de couple (`cons`) de la forme `(clé . valeur)`. Écrire une fonction `int nombre_entier(sexp env)` qui parcourt l'environnement donné en paramètre et renvoie le nombre d'entiers parmi les valeurs.

```
int nombre_entier (sexp env) {
    sexpr e, couple;
    int n = 0;
    for (e=env; e!=NULL; e=cdr(e)) {
        couple = car(e);
        valeur = cdr(couple);
        if (integer_p(valeur)) n++;
    }
    return n;
}
```

2. On cherche maintenant à travailler sur l'allocateur. Écrire une fonction `int plus_grand_espace_libre()` qui parcourt la double liste chaînée des blocs de `MEMOIRE_DYNAMIQUE` et renvoie l'indice du plus grand bloc libre de la mémoire.

```
int plus_grand_espace_libre() {
    int i;
    int m = 0;
    for (i=0; i != suivant(i); i = suivant(i)) {
        if (taille(i) > taille(m) && est_libre(i)) {
            m = i;
        }
    }
    if (m==0 & !est_libre(0)) return -1;
    return m;
}
```

3. Écrire une fonction `int agrandir_basique(int i)` qui fusionne le bloc `i` avec le bloc suivant s'il est libre, ou qui ne fait rien sinon. Dans tous les cas, on renverra la taille du bloc `i`.

```
int agrandir_basique(int i) {
    int p, j, k;
    if (est_libre(suivant(i))) {
        /* AVANT : p -> i -> j -> k  -> l */
        /* APRÈS : p -> i ------> k  -> l */
        p = precedant(i);
        j = suivant(i);
        k = suivant(j);
        MEMOIRE_DYNAMIQUE[i] = cons(0,p,1,k);
        MEMOIRE_DYNAMIQUE[k] = cons(0,i,1,l);
    }
    return taille(i);
}
```

4. Écrire une fonction `int agrandir(int i, int nouvelle_taille)` qui correspond grosso-modo à la fonction `realloc`. La variable `nouvelle_taille` est exprimé en nombre de blocs et non en nombre d'octets. La fonction essaiera d'agrandir le bloc `i` s'il y a assez de place disponible à la suite du bloc d'indice `i`. Sinon, elle cherchera dans la mémoire un espace suffisamment grand pour allouer un nouveau bloc, copiera le contenu de l'ancien bloc vers le nouveau et libérera l'ancien. Si la réallocation n'a pas fonctionnée, on renverra `-1`, sinon, on renverra un pointeur vers le bloc agrandi.

```
int agrandir(int i, int nouvelle_taille) {









}
```